COMPLETE EMBEDDED SOLUTIONS

12276 San Jose Blvd #119 • Jacksonville, FL 32223
Phone: (904) 880-1840 • Fax: (904) 880-1632
Email: cmx@cmx.com • WWW: http://www.cmx.com

CMX Systems, Inc. CMX Scheduler for dsPIC(tm) MCUs Software License Agreement

BY USING THE CMX- SCHEDULER SOFTWARE (THE "PRODUCT"), YOU ARE CONSENTING TO BE BOUND BY AND BECOME A PARTY TO THIS AGREEMENT AS THE "LICENSEE." IF YOU DO NOT AGREE TO ALL OF THE TERMS OF THIS AGREEMENT, YOU MUST NOT INSTALL OR USE THE PRODUCT, AND YOU DO NOT BECOME A LICENSEE UNDER THIS AGREEMENT.

1. License Agreement. As used in this Agreement, "CMX" shall mean CMX Systems, Inc. CMX grants Licensee a non-exclusive and non-transferable license to use the object code version of the Product. This license does not entitle Licensee to receive from CMX hard-copy documentation, technical support, telephone assistance, or enhancements or updates to the Product. CMX may terminate this Agreement if Licensee breaches any of its terms and conditions. Upon termination, Licensee shall destroy all copies of the Product.

2. Restrictions. Without CMX's prior written consent, Licensee may not: (i) modify or create any derivative works of the Product or documentation, including customization, translation or localization; (ii) decompile, disassemble, reverse engineer, or otherwise attempt to derive the source code for the Product (iii) redistribute, encumber, sell, rent, lease, sublicense, or otherwise transfer rights to the Product; (iv) remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels in the Product; or (v) publish any results of benchmark tests run on the Product to a third party. Licensee's use of the Product shall be limited to integrating the Product as an integral component within the Licensee's own product or products. Licensee shall have the right to distribute the Product as an integral component of the Licensee's own products as long as the Product is in absolute machine readable format (e.g., HEX file). The Licensee cannot sell a product that allows the user(s) of the Licensee product(s) to be able to indirectly call the CMX functions (e.g., The Licensee product contains an API [Application Program Interface] that allows the user(s) to indirectly call and use the Product's functions.)

3. Fees. There is no license fee for the Product. If Licensee wishes to receive the Product on media, there may be a small charge for the media and for shipping and handling. Licensee is responsible for any and all taxes.

4. Proprietary Rights. Title, ownership rights, and intellectual property rights in the Product shall remain in CMX. The Product is protected by copyright and other intellectual property laws and by international treaties.

5. Disclaimer of Warranty. THE PRODUCT IS PROVIDED FREE OF CHARGE, AND, THEREFORE, ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION THE WARRANTIES THAT IT IS FREE OF DEFECTS, VIRUS FREE, ABLE TO OPERATE ON AN UNINTERRUPTED BASIS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT. NO USE OF THE PRODUCT IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER
.

6. Limitation of Liability. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL CMX OR ITS AFFILIATES, DISTRIBUTORS (NOT LIMITED TO MICROCHIP), OR SUPPLIERS BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE PRODUCT, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF, AND REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED. IN ANY CASE, THE COLLECTIVE LIABILITY OF CMX, ITS AFFILIATES, DISTRIBUTORS (NOT LIMITED TO MICROCHIP), OR SUPPLIERS UNDER ANY PROVISION OF THIS AGREEMENT SHALL NOT EXCEED IN THE AGGREGATE THE SUM OF THE FEES LICENSEE PAID FOR THIS LICENSE (IF ANY). SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL, CONSEQUENTIAL OR SPECIAL DAMAGES, SO THIS EXCLUSION AND LIMITATION MAY NOT APPLY TO YOU.

7. Export Control. Licensee agrees to comply with all export laws and restrictions and regulations of the United States or foreign agencies or authorities, and not to export or re-export the Product or any direct product thereof in violation of any such restrictions, laws or regulations, or without all necessary approvals. As applicable, each party shall obtain and bear all expenses relating to any necessary licenses and/or exemptions with respect to its own export of the Product from the U.S. By installing or using the Product, Licensee agrees to the foregoing and represents and warrants that it complies with these conditions.

8. U.S. Government End-Users. The Product is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End-Users acquire the Product with only those rights set forth herein.

9. Miscellaneous. This Agreement constitutes the entire agreement between the parties concerning the subject matter hereof, and may be amended only by a writing signed by both parties. This Agreement shall be governed by the laws of the State of Florida, U.S.A. All disputes relating to this Agreement are subject to the exclusive jurisdiction of the courts of Florida and you expressly consent to the exercise of personal jurisdiction in the courts of Florida in connection with any such dispute including any claim involving CMX. This Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods. If any provision in this Agreement should be held illegal or unenforceable by a court of competent jurisdiction, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or severed from this Agreement if no such modification is possible, and other provisions of this Agreement shall remain in full force and effect. A waiver by either party of any term or condition of this Agreement or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof. If any dispute arises under this Agreement, the prevailing party shall be reimbursed by the other party for any and all legal fees and costs associated therewith.

dsPIC(tm) is a trademark of Microchip Technology Inc. and used with permission.

12276 San Jose Blvd #119 • Jacksonville, FL 32223
Phone: (904) 880-1840 • Fax: (904) 880-1632
Email: cmx@cmx.com • WWW: http://www.cmx.com

SOFTWARE PROBLEM REPORTING

If the user feels there is a problem with the CMX software, the following steps should be performed.

Please be some what ensured that it is a problem with the CMX Real-Time Multi-Tasking operating System code supplied and NOT: application coding errors, poor design of the application code and how it is integrated, misuse of the CMX RTOS, etc. Try a different approach or try to exercise just the particular problem in a minimal test if possible.

Email ONLY, the following.

PRODUCT DETAILS: The company name, name of person reporting problem, the name and version of the manufacturer's C compiler being used, the version number and version number of the CMX Scheduler RTOS.

PROBLEM SEVERITY: how serious is the problem.

PROBLEM DESCRIPTION: a concise and informative description of the problem, the results that occurred, and any solution if known.

CMX will try to resolve any problems within a reasonable amount of time. If needed, CMX may request the user to send or fax the portion of the code that is not working correctly.

DISCLAIMER: While CMX Systems, Inc. will investigate the problem as soon as possible and make every attempt to come up with a solution, we do not guarantee to provide a solution.

NOTICE: CMX is not in the business of writing other companies application code around the CMX Real-Time Multi-Tasking Operating System, so DO NOT email us expecting this. Also if the user manipulates the CMX code for any reason, then CMX will not be responsible for how the code executes.

Software updates will be posted on our website for users to download. CMX will NOT notify you, since this is a free RTOS. We recommend that you check our website every so often to check to see if a new updated version is present.

USER MANUAL

Copyright© 2002
All rights Reserved

CMX Systems, Inc.
12276 San Jose Blvd #119
Jacksonville, FL 32223 U.S.A.
Phone: (904) 880-1840
FAX: (904) 880-1632
Email: cmx@cmx.com
WWW: http://www.cmx.com

IMPORTANT LEGAL NOTICE

Please note that there are specific references made in this user's manual, in our marketing literature, and in the actual software files to CMX Systems, Inc., and CMX Company. CMX Company is a division of CMX Systems, Inc. and the two company names should be regarded as the same legal entity for the purposes of copyright, trademark, licensing and any other legal issues.

COPYRIGHT NOTICE

Copyright© 2002 CMX Systems, Inc. All rights reserved.

No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of CMX Systems, Inc.

This documentation is confidential. CMX may, if they so choose, to put some portions of this documentation within the public domain and if so, then those respective portions will not be consider confidential.

DISCLAIMER

The information in this document is subject to change without notice and does not represent a commitment on any part of CMX Systems, Inc. While the information contained herein is assumed to be accurate, CMX Systems, Inc. assumes no responsibility for any errors or omissions.

In no event shall CMX Systems, Inc., its employees, its contractors, or the authors of this document be liable for special, direct, indirect, or consequential damage, losses, costs, charges, claims, demands, claim for lost profits, fees, or expenses of any nature or kind.

TRADEMARKS

CMX is a trademark of CMX Systems, Inc.

All other product names are trademarks or registered trademarks of their respective owners.

Table of Contents

The CMX Scheduler Multi-Tasking Executive 1
- Getting started 2
- The CMX Scheduler 4
- When A Task Is Interrupted 4
- Task States 5
- Setting Up Tasks 6
- CMX Scheduler task control blocks 6
- CMX Return Status Byte Values 7
- CMX DATA TYPES 7
- Layout Of The Functions 8
- Task Manager Functions 9
  - The K_Task_Create function 10
  - The K_Task_Start function 12
  - The K_Task_Wait function 13
  - The K_Task_Wake function 15
  - The K_Task_Coop_Sched function 16
  - The K_Task_End function 17
  - The K_Task_Kill function 18
- Event manager functions 20
  - The K_Event_Wait function 20
  - The K_Event_Signal function 23
  - The K_Event_Reset function 26
- Operating System Functions 27
  - The K_OS_Init function 28
  - The K_OS_Start function 28
  - The K_OS_Intrp_Entry function 29
  - The K_OS_Intrp_Exit function 30
  - The K_OS_Tick_Update function 30
  - The K_OS_Enable_Interrupts function 32
  - The K_OS_Disable_Interrupts function 33
- processor specific chapter 34
- Configuring the RTOS 38
- Stacks In General 40
- The CMX Scheduler Chapter 44

QUICK REFERENCE 47
- CMX Return Status Byte Values 47
  - K_Event_Reset 48
  - K_Event_Signal 50
- Event Manager Function 52
  - K_Event_Wait 52
  - K_OS_Disable_Interrupts 54
  - K_OS_Enable_Interrupts 56
  - K_OS_Init 57

K_OS_Intrp_Entry....58
K_OS_Intrp_Exit....59
K_OS_Start....60
K_OS_Tick_Update....61
K_Task_Coop_Sched....63
K_Task_Create....64
K_Task_End....66
K_Task_Kill....68
K_Task_Start....70
K_Task_Wait....72
K_Task_Wake....74

THE CMX SCHEDULER MULTI-TASKING EXECUTIVE

The CMX Scheduler real time multi-tasking operating system is designed for the Microchip dsPic processor.

The CMX Scheduler RTOS has fewer functions supplied than the other two CMX RTOSes called CMX-TINY+ and CMX-RTX. The CMX Scheduler is designed to be upward compatible to CMX-TINY+ RTOS. It is also upward compatible to CMX-RTX though there are some differences in some of the function prototypes and what is needed for parameters.

The purpose of the CMX Scheduler version is that it is free (or a very minimal cost) and to provide a true preemptive operating system, with a number of useful functions and to use as LITTLE ROM and RAM as possible. There is a limit to the number of tasks and that limit is set to 5. Users that need more functionality and/or more tasks can purchase the CMX-TINY+ or CMX-RTX RTOS.

The CMX Scheduler uses very minimal ROM and RAM. The user may write his program in C, assembly or a combination. No source code is provided for the CMX Scheduler (source code is supplied with CMX-TINY+ and CMX-RTX RTOSes).

There is one CMX functions (K_Event_Signal) that has an extra parameter that is NOT used now, but for upward compatibility to CMX-TINY+, CMX decided to leave it in. It is the 'mode' parameter and it should be set to zero (0). For those of you that want to, you can download our CMX-TINY+ manual from our website, to see what the mode parameter will do in the CMX-TINY+ RTOS. It really is not even passed to the function, but must be there.

Welcome to the world of multi-tasking. CMX provides the necessary function calls and operating system to write efficient C and/or assembly code to create a well designed, multi-tasking application.

The ability of a single processor to run many tasks by swapping different tasks in and out creates the feeling that many tasks are operating simultaneously. This is what multi-tasking is all about. CMX Scheduler is a real-time multi-tasking operating system giving you function calls and an operating system kernel that will:

allow control of tasks
handle events
regulate timing in a variety of ways
swap tasks and interrupts.

GETTING STARTED

CMX highly recommends you read the complete manual, the processor specific section and any additional CMXREAD.DOC files that may be supplied before trying to interface with CMX. You should realize there will be a “learning curve” as with any new software. The more you work with it, the better you will understand the CMX software and how to incorporate it into your application code.

The manual is written for those coding in C. To code in assembly is very easy, just use the following rules. There is a macro file that should be brought into any assembly coded file, named CMXMACRO.INC. The CMX functions are type in (through any editor) just as you would type them in C. If there is one or more parameter(s) needed, you pass them in the same order as they would be passed in C. Note that you do not put parenthesis around the parameters. Below is an example, one code for C, one code for assembly.

C example:

```
void task1(void)
{
   K_Event_Wait(0x01,20,2);
}
```

Assembly example:

```
.include "cmxmacro.inc"

.text
_task1:
   K_Event_Wait 0x01,20,2
```

The CMXMACRO.INC will automatically put the parameters in the proper registers and call the specified function.

You will need the following CMX files in your project.

CMX_INIS.O - CMX initialization module

LBDSPIC.A - CMX library of CMX functions.

☞*For the next 2 listed, one or the other is needed depending upon the usage of the MAC unit. Do NOT include both*

CSSKV1.O - For those users NOT using the MAC unit, meaning the ACCA and ACCB are not being used.

CSSKV1A.O - For those users USING the MAC unit, meaning the ACCA and ACCB are being used.

CMX also includes to sample programs, one written in C and another written in assembly. The C project is called TINYS.MCW and the assembly project is called TINYSASM.MCW

Please copy the csfunc.h header file to the INCLUDE directory of Microchip C tools.

When creating a project, you will need to bring in the CMX files listed above, your modules and also the proper Microchip achive (library files) that are needed. Please read more about how to create a project within the Microchip tools documentation

THE CMX SCHEDULER

The CMX Scheduler operating system provides real-time processes the ability for task switching according to the input stimulus received. The heart of this operating system is the scheduler. The scheduler is based on true preemption. This means that tasks and interrupts can cause an immediate task switch, if a higher priority task becomes able to run as the result of a CMX function. Cooperative scheduling is also possible, a task can let the next task (same or lower priority) run if desired.

The scheduler keeps track of variables associated with tasks. Task switches will be performed by the scheduler depending upon the state of these variables. A task or interrupt may cause a preemption, which informs the scheduler that a higher priority task needs servicing. Possibly the interrupt that determines the “system tick”, the basis for all time related activity, occurred. The CMX tick function will determine whether there are any time related activities that need attention and will handle the necessary timing chores.

If a preemption occurs, the task that was running has its context saved. The scheduler decides which of the tasks to run next. The scheduler will load all the proper information for this task to operate. If the task had been suspended, for whatever reason, by a function call or a higher priority task, then all the saved task variables are reloaded, all registers are restored to their respective values for this task and the task resumes where it left off as if it was never suspended.

For more information, please study the CMX Scheduler Chapter which explains how the scheduler works in detail. That chapter explains the different flags the scheduler will act upon and how the scheduler interfaces to the interrupt driven *K_OS_Tick_Update* function, tasks, CMX functions and other interrupts.

WHEN A TASK IS INTERRUPTED

A task may suspend itself by a variety of CMX function calls forcing the scheduler to reschedule immediately without regard to the specified system tick time interval. Also some CMX function calls that take a task out of the suspended state, or that start a task that was idle, will force immediate rescheduling, if the suspended or new task has a higher priority than the running task.

CMX gives you the ability to handle single or multiple (nested) interrupts. The interrupts may call many CMX functions and CMX provides the necessary interrupt functions that save and restore the contexts of a task, or interrupt, when nesting occurs.

When a task's (or interrupt's) context is saved, you can be assured that all parameters dealing with this task or interrupt -- all the CPU registers, local variables and parameters passing variables -- will be restored properly, as if the task or interrupt was never suspended.

TASK STATES

There are several possible states a task can be in, though a task can be in only one state at a time. These states are: IDLE, READY, RUN, WAIT, ready to RESUME and KILL.

IDLE state

A task that has been created with the *K_Task_Create* function, but not started by the *K_Task_Start* function is in the IDLE state. A task that has completed its code and called the *K_Task_End* function is placed into the IDLE state if there are no outstanding triggers in its control block. A task that is in its IDLE state will not run.

READY state

The READY state informs the scheduler that the task is ready to run, but NOT running. This allows the scheduler to determine what task to run when a scheduling takes place according to the task's priority in relation to the other tasks' priorities.

RUN state

The task that is executing is in the RUN state and owns the CPU time. Only one task may be in the RUN state at any one time.

WAIT (suspended) state

A task that suspended itself by a CMX function call is in the WAIT (suspended) state. There are many function calls that will suspend a task. The WAIT state consists of a task that is waiting on one or more of the following: time, events, flags, messages, on a reply, etc.

RESUME state

The RESUME state is treated the same as the READY state. The only difference is that it informs the scheduler that the task had been started, yet not finished, with its code. This means that a higher priority task has preempted and forced the original task that was running to become ready to RESUME, or that the task had suspended itself by a function call and now has removed itself out of the WAIT (suspended) state into the ready to RESUME state.

KILL state

A task that is KILLED (removed) has its state put into the KILL state, indicating this task no longer exists.

SETTING UP TASKS

Tasks should be coded to perform specific duties. It is up to you to properly create the tasks to do specific jobs in an orderly fashion. Structuring the individual tasks' responsibilities in relation to each other, and providing the proper interrupt handling with respect to other interrupts and tasks is probably the most challenging problem in writing real-time multi-tasking code.

CMX SCHEDULER TASK CONTROL BLOCKS

There are two task control blocks which contain certain information about a task. The first control block (ROM TCB) normally resides in the text area (text, code, ROM, etc., area). This task control block identifies a task's starting address, the task's system stack starting address and its priority. Because most processors with limited amount of RAM have enough ROM, this is where some of the information resides for the tasks in a particular application.

The next task control block (RAM TCB) resides in RAM located on the processor's silicon. The RAM TCB contains the state of the task, the task timer, which is used by the RTOS functions when a time-out period is specified, the address of the task's saved stack, the task event bit and finally the task's position in the other task control block (ROM TCB).

The ROM TCB looks as follows:

```
typedef struct {
    void (*task_addr)(void);            /* Task's CODE address.  */
    unsigned short *system_stack;    /* Task's stack start address.  */
    unsigned char init_priority;       /* Task's priority.  */
    } ROM_TCB;
```

An example of the ROM_TCB structure:

```
unsigned short task_stack[200];       /* global array for task stacks.  */

ROM_TCB task_rom_list[]= {

    /* The first entry MUST be declared as follows, used by the RTOS.  */
    main, 0, 0,
    task1, &task_stack[0], 5,
    task2, &task_stack[50], 3,
    task3, &task_stack[100], 6,
    task4, &task_stack[150], 2
    };
```

The first task within the ROM_TCB must be the the proram's *main function*. The *main* function uses the startup stack, so it is not necessary to specify a stack size for it. The main function is really NOT a task, but is needed here. The stack for each task is assumed to grow UPWARDS (From low to high memory). The size of a particular task stack must be large enough to hold the following: the registers, the number of return addresses (generated by calls to functions), the number of local variables within the functions called and the number of arguments placed on the stack by a calling function. You MUST be careful that the size of the task stack is large enough. Failure to do so will result in system crashes. We recommend that the initial stack size be doubled over the estimated size until testing shows the size can be reduced.

As you can see, there are 4 user tasks in the ROM TCB. The *main function MUST* be the first task listed.

Time slicing is not available for the CMX Scheduler RTOS as it is in the full-featured CMX-RTX RTOS. This is because more RAM would be required to support this and CMX believes time slicing is not needed in most applications.

CMX RETURN STATUS BYTE VALUES

Symbol	Hex	Value	Explanation
K_OK	00	Good	CMX call was successful
K_TIMEOUT	01	Warning / Error	Time out occurred
K_NOT_WAITING	02	Error	Task not waiting for wake request
K_ERROR	FF	Error	General error, CMX call unsuccessful

In some cases a return value of zero indicates the CMX function is telling the caller the item it wanted was not there or a time out occurred.

CMX DATA TYPES

CMX has declared the following data type names within the cxdefine.h header file. There are a few listed here. These may possibly change depending on the processor and C vendor that you are working with. Look at the cxdefine.h file, to see the ones pertinent to your processor and C vendor.

```
#define byte unsigned char
#define word16 unsigned short
#define bit_word16 unsigned short or unsigned int
#define sign_word16 signed short
#define word32 unsigned long
```

LAYOUT OF THE FUNCTIONS

Within each function, we will describe in detail the purpose of that particular function. Explained will be the parameter declarations, if any, the functions's return type, if any and any other useful information. Also within each function section, will be the following:

Called

Before entering RTOS, Tasks, Interrupts.

[Before entering RTOS] means the call may be used prior to entering the CMX operating system. [Tasks] mean a task may use this CMX call. [Interrupts] mean interrupts may use this CMX call.

☞ *The K_OS_Init function must be called before any CMX function call may be used.*

The user will then see what header files are needed that identify the function prototypes. Also the parameters needed will be shown in an example style, with a brief commented description, followed by the function prototype, with the respective example parameters used within the function prototype. An example is below.

#include <csfuncs.h> /* has function prototype */

Many of the parameters passed are of constant value, so we have used #defines to identify them, with 3 question marks (i.e. ???), indicating the value would need to be selected by the user. Also we have tried to use descriptive words to indicate the meaning of the particular parameter. You may choose any text you like for the name of a parameter, but we recommend that it be meaningful.

#define TASK1_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Create*(TASK1_ID);

Passed

This will identify each of the parameters that this function will be passed and what they indicate.

TASK1_ID is the index into the ROM TCB array for the task.

Returned

This will identify the return-type of this function and what it indicates.

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: ROM_TCB number out of range.

If STATUS equals K_OK, then the parameters contained within the ROM TCB are used to set up the ROM TCB array. The task is put in the IDLE state and must be triggered with the *K_Task_Start* function to enable this task to be READY to run.

Example

We will then provide one or more examples of using the function and a comment on how we are using it.

The following is an example for creating a task to the RTOS.

void task1(void); /* function prototype, show that task1 does not receive nor return parameters */

#define TASK1_ID 1

```
void main(void)
{
   unsigned char status; /* create a local status byte */
   status = K_Task_Create(TASK1_ID);
   /* call CMX function K_Task_Create to create task1 */
   if (status != K_OK) /* check status, make sure good function
   call */
      {
      error_handler(); /* go to error handler */
      }
}
```

There may or may not be an additional comment field as shown below. The comment field is to reinforce what we are trying to do with the function.

Comments

CMX returns the status of the *K_Task_Create* function call indicating whether the call was successful or not. If STATUS equals K_OK, then the parameters contained within the ROM TCB are used to set up the ROM TCB array. The task is put in the IDLE state and must be triggered with the *K_Task_Start* function to enable this task to be READY to run.

TASK MANAGER FUNCTIONS

The task manager is part of the CMX library and provides the necessary functions for controlling your tasks. The task manager functions are listed below along with their reference pages.

K_Task_Create (Page 10)
K_Task_Start (Page 12)
K_Task_Wait (Page 13)
K_Task_Wake (Page 15)
K_Task_Coop_Sched (Page 16)
K_Task_End (Page 17)
K_Task_Kill (Page 18)

The K_Task_Create function

The CMX *K_Task_Create* function is used to create a task. You can create a task before entering the CMX operating system or dynamically while running under the CMX operating system. For maximum speed and because you may know ahead of time what tasks will be needed, it is highly recommended that the tasks be created before entering the RTOS. The creation identifies to the ROM TCB which ROM TCB task now owns the index slot of the ROM TCB. It also places the task into the IDLE state.

The ROM TCB index is the only parameter that is sent to the *K_Task_Create* function. This is the index into the ROM TCB structure array. The *K_Task_Create* function then uses the necessary information from the ROM TCB and determines where the task will be placed within the ROM TCB.

The priority for this task is located in the ROM TCB array. The lower the priority number, the higher the priority of the task.

The *K_Task_Create* function returns the code K_OK if the function was successful in creating the task, or the general error code (K_ERROR, value of 0xFF) if it was NOT created. The K_ERROR code indicates the task's ROM TCB index number was not correct or out of range.

This is an example of the *K_Task_Create* function:

Called

Before entering RTOS, Tasks.

```
#include <csfuncs.h> /* has function prototype */
byte K_Task_Create(byte); /* this is the function prototype */

#define ROM_TCB_ID ??? /* index into the ROM_TCB array where this task resides.
*/

unsigned char STATUS; /* should be local */

STATUS = K_Task_Create(ROM_TCB_ID);
```

Passed

ROM_TCB_ID is the index number into the ROM_TCB array that describes this task.

Returned

CMX returns the status of the *K_Task_Create* function call indicating whether the call was successful or not. If STATUS equals K_OK, then the parameters contained within the ROM TCB are used to set up the ROM TCB array. The task is put in the IDLE state and must be triggered with the *K_Task_Start* function to enable this task to be READY to run.

Example

The following is an example for creating a task for the RTOS.

void task1(void); /* function prototype, show that task1 does not receive or return parameters */

#define TASK1_ROM_TCB_ID 1 /* task 1 ROM TCB index. */

#define TASK2_ROM_TCB_ID 2 /* task 2 ROM TCB index. */

```
void main(void)
{
   unsigned char STATUS;
   STATUS = K_Task_Create(TASK1_ROM_TCB_ID);
   /* call the RTOS function K_Task_Create with the index into
   the ROM TCB for task 1. */
   if (STATUS == K_ERROR)
      {
      /* Error, handle it here */
      }

   STATUS = K_Task_Create(TASK2_ROM_TCB_ID);
   /* call the RTOS function K_Task_Create with the index into
   the ROM TCB for task 2. */
   if (STATUS == K_ERROR)
      {
      /* Error, handle it here */
      }
}
```

Comments

We are creating task 1. Tasks may not return or receive arguments. Tasks parameters, such as the task's address, initial priority, stack address, etc. are located in the ROM TCB array.

The K_Task_Start function

Since the *K_Task_Create* function puts a task into the IDLE state when created, this function allows a task to be started. This function may be called anytime. The *K_Task_Start* function really puts the task into the READY state allowing it to become the RUNNING task when it is the highest priority task ready to run. If the task was IDLE, then the task becomes READY, however not necessarily the RUNNING task. Once a task is out of the IDLE state any additional *K_Task_Start* calls to this task are ignored.

When the task normally ends its code and puts itself back into the IDLE state, if there are any outstanding start requests then the task will automatically put itself back into the READY state. The task's ROM TCB index number is passed to this function indicating which task to start. This function can be called before entering the CMX operating system and while in the operating system.

This is an example of the *K_Task_Start* function:

Called

Before entering RTOS, and tasks.

#include <csfuncs.h> /* has function prototype */
byte *K_Task_Start*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Start*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to start.

#define TASK1_ROM_TCB_ID 1 /* task1 ROM TCB index number. */

```
void task2(void)
{
   unsigned char status;
   status = K_Task_Start(TASK1_ROM_TCB_ID); /* start task 1, put
   into READY state if this is the first trigger */
   if (status != K_OK) /* check status, make sure good function
   call */
      {
      error_handler(); /* go to error handler */
      }
}
```

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the ROM TCB index number does not exist.

If STATUS equals K_OK, then the task is put into the READY state. If it is in the READY state already, then no further action will occur.

Again the status is passed back indicating whether the CMX *K_Task_Start* call was successful or not. If the task being started has a higher priority (lower priority number) than the task that is currently RUNNING, then an immediate task switch will occur, bypassing the normal rescheduling caused by the system tick.

The K_Task_Wait function

The *K_Task_Wait* function enables a task to suspend itself for a specified amount of time or indefinitely. This function allows a task to become synchronized with another task, interrupts, or the system tick. Letting the task suspend itself for a specified period, knowing that the task will RESUME when the time expires, can be very useful. This function also allows a task to wait with or without a time period for another task to wake it, notifying the task that an event has happened. (The event manager handles multiple events per task. This is described fully later). Only tasks may call this function.

The amount of time, which is a multiple of system ticks, is passed to this function. This value is the number of system ticks you want the task suspended for. The amount of time may be from zero to 65535 (zero to FFFF hex). A time period of zero will result in the task waiting indefinitely until the *K_Task_Wake* function is used to wake this task.

The task will automatically suspend itself for a specified time period (if non zero) and then become READY to RESUME execution at the end of this time. The *K_Task_Wake* function may be called to wake this task earlier than the time specified. This will be reported by a returned status byte. Remember that other tasks may use the *K_Task_Wake* function to wake this task.

The accuracy of this call is a derivative of the system tick specified. For example, say you have created a 20 millisecond system tick. If the task requests 10 system ticks, and calls *K_Task_Wait*, the following will happen. The time period is decremented at every system tick. When the time becomes zero, then the task is automatically put into the READY to RESUME execution state. You can see, depending on when the task calls *K_Task_Wait* in relation to the system tick, the task will wait anywhere from 180 milliseconds to 200 milliseconds.

This is an example of the *K_Task_Wait* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
byte *K_Task_Wait*(word16); /* this is the function prototype */

#define TIME_CNT ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wait*(TIME_CNT);

Passed

TIME_CNT is number of system ticks that this task will suspend itself. If the value is zero then the task will be suspended indefinitely until the *K_Task_Wake* function is used. If the value is non-zero, then the task will be suspended for that number of system ticks. The *K_Task_Wake* function may be used prior to the time period expiring, to wake this function and put it back into the READY state. The maximum value that TIME_CNT may be is 65535.

```
void task2(void)
{
   unsigned char status;
   status = K_Task_Wait(100); /* set task2 to wait for 100 system
   ticks */
   if (status != K_OK) /* check status, make sure task was woken
   up before time elapsing */
      {
      /* maybe take corrective action if the time period expired,
      unless the task wanted to be synchronous with the system
      tick */
      }
}
```

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_TIMEOUT = Warning: the time specified has elapsed.

If STATUS equals K_OK, then the task has RESUMED execution because the *K_Task_Wake* function was used to wake this task. If STATUS equals K_TIMEOUT, then the time period specified has expired and this is why the task was awakened.

> ✗ **WARNING: This function does not test to see if the caller is a task or not, so make sure that only tasks call this function.**

The status returned by this function will indicate that either the time period specified (if non zero) has elapsed, or that the task was awakened before the time period specified by the *K_Task_Wake* function.

The K_Task_Wake function

The *K_Task_Wake* function wakes a task that had put itself into the suspended state. Tasks may call this function. The caller sends the task's ROM TCB index number to the *K_Task_Wake* function. This function then takes the task and places it into the ready to RESUME execution state.

Note that the awakened task might not immediately become the RUNNING task because of its priority. If the task's priority is higher than the current RUNNING task, then an immediate task switch will occur regardless of the system tick.

This is an example of the *K_Task_Wake* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
byte *K_Task_Wake*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wake*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to wake up.

#define TASK1_ROM_TCB_ID 0

```
void task2(void)
{
   unsigned char status;
   status = K_Task_Wake(TASK!_ROM_TCB_ID); /* wake task 1 up */
```

```
   if (status != K_OK) /* check status, make sure task was
   waiting */
      {
      /* maybe take corrective action if task 1 wasn't waiting */
      }
}
```

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task ROM TCB index number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

The K_Task_Coop_Sched function

The *K_Task_Coop_Sched* function does a cooperative rescheduling. Normally CMX reschedules when a higher priority task is READY to run. If a task calls this function then CMX schedules the next task that is READY to run despite its priority.

This allows a task to let another task of the same or lower priority become the RUNNING task (if the task is READY). The task that calls this function will immediately be placed into the ready to RESUME state. In most cases, this function would not be used because the operating system is based on preemption (meaning the highest priority task that can run is the RUNNING task). The task could call the *K_Task_Wait* function specifying a time out period of one, which would let another task with the same priority as the calling task become the RUNNING task. Only tasks should call this function.

This is an example of the *K_Task_Coop_Sched* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
void *K_Task_Coop_Sched*(void); /* this is the function prototype */

Passed

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
   /* application code here */
   K_Task_Coop_Sched(); /* allow a rescheduling now, do not wait
   for normal preemption to perform a normal scheduling, NOTE:
   task context is saved completely and will RESUME when task is
   highest priority READY to run */
}
```

Returned

No status is returned.

The K_Task_End function

This function allows a task to terminate itself either prematurely or at the end of its code. The *K_Task_End* function MUST be called by all tasks that normally would hit their “end brace”. If a task calls this function before its end brace, then all variables, pushes and calls on the stack will be forgotten. This function is also useful for exiting out of a serious or nonrecoverable error. The task could first pass a message to another task with the message saying what task and what type of error and then call *K_Task_End*.

When the *K_Task_End* function is called, the task is automatically terminated, resetting its stack pointer and code pointer to the task's beginning. The task is still able to execute whenever it is started again. Only tasks should call this function.

☞ If, because of an indefinite while statement, the task never hits its end brace, then the task does not need the K_Task_End function call at the task's end brace.

This is an example of the *K_Task_End* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
void *K_Task_End*(void); /* this is the function prototype */

Passed

Nothing is passed. Only tasks can call this function.

```
void task2(void)
{
   ..../* application code here */
   K_Task_End(); /* MUST be the last C statement before the
   task's end brace */
}
```

OR

```
void task2(void)
{
   ..../* application code here */
   if (???) /* serious error or non recoverable */
      {
      K_Task_End(); /* End task 2, because of serious or non-
      recoverable error */
      }
   ..../* more code here */
   K_Task_End(); /* normal exit function here */
}
```

Or if a task is never going to hit its end brace, such as below.

```
void task2(void)
{
   while(1)
      {
      ..../* application code here */
      }
   /* K_Task_End function NOT NEEDED here */
}
```

Returned

No status is returned.

☞ *An immediate rescheduling (with possibly this task executing from its beginning brace) will occur when this function is called. If the task has additional trigger (K_Task_Start) requests, then the task will be put into the READY state, otherwise the task will become IDLE. Remember, all local variables will be lost. Other tasks may be waiting for this task to use a CMX function call to wake it.*

✗	**WARNING: ALL tasks must call this function prior to executing their right-end brace. If the task will never, for whatever reason, execute its end brace, then this function does not have to be called. It is highly recommended that all tasks have this function.**

The K_Task_Kill function

This function gives you the ability to remove a task permanently from the RAM TCB task control block queue. The calling task will send the ROM TCB index number of the task to be removed (it may send its own ROM TCB index number if desired) to the *K_Task_Kill* function.

If the task calling the *K_Task_Kill* function is trying to remove itself, then it will be removed and an immediate task switch will occur. Please note: all tasks waiting on the removed task will not be notified when the task is removed and may wait forever.

This is an example of the *K_Task_Kill* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
byte *K_Task_Kill*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Kill*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to kill.

#define TASK1_ROM_TCB_ID 0

```
void task2(void)
{
   /* application code here */
   K_Task_Kill(TASK1_ROM_TCB_ID); /* remove task1, any further
   reference to task1 will be in error */
}
```

Returned

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task ROM TCB index number does not exist.

If STATUS equals K_OK, then the task has been removed successfully. If the task is removing itself, then an immediate task switch will occur.

EVENT MANAGER FUNCTIONS

The event manager is part of the CMX library and provides the necessary functions for controlling events. The event manager functions are listed below along with their page references.

K_Event_Wait (Page 20)
K_Event_Signal (Page 23)
K_Event_Reset (Page 26)

The CMX event functions are very sophisticated and powerful, yet easy to use. Each task has a SINGLE event. Each event is considered a bit, with the bit capable of being either set or cleared. An event, or bit, that is set indicates the event occurred. Tasks can wait on this event. When the event the task is waiting on occurs, the task is awakened and notified that the event occurred. Also, a task can specify a timeout period to wait for, or to wait indefinitely.

The task can specify the event it is waiting for be automatically cleared prior to waiting for that event (which means the task will become suspended), automatically cleared after the event occurred, and the task is awakened, or not cleared at all. Tasks also can call a function that will reset an event at any time.

The K_Event_Wait function

This function allows a task to wait for an event with a specified time out if so desired. This function is very flexible and powerful because it allows a task to wait for an event to happen. It also allows the task to specify a certain amount of time it is willing to wait, so it can take corrective action if the event does not become set. Also, the clear mode command offers the task the ability to be truly synchronized or not with respect to the entities that may set an event such as other tasks and interrupts.

The task supplies the *K_Event_Wait* function with three parameters:

1.) the event to wait on

2.) the command whether to automatically clear the event the task wants to wait on prior to testing, after the event happen, both prior and after or not at all

3.) A time period indicating whether to wait for a certain amount of time or indefinitely for a match. A time period of zero means that the task will wait indefinitely for the event(s) to occur.

The specified event are tested when this function is called, if autoclear prior to testing is not specified. If the event specified is set when the *K_Event_Wait* function is called, the task is returned to immediately identify the event is set. If the event is not present when the task calls this function, then the task will wait for this event. The task will automatically resume (wake up) when specified event is set or the specified time period expires.

The clear mode command

The clear mode command has four different possibilities.

A clear mode value of 0 (zero) indicates not to clear the task's event flag at all when the *K_Event_Wait* is called and used. This means the task's event will stay in it's present state.

A clear mode value of 1 indicates the *K_Event_Wait* function will automatically clear the task's event it is waiting on. This means if the task is about to wait for an event, then that event will be cleared prior to testing to see if there is a match. This will result in the task becoming suspended for there will be no match.

A clear mode value of 2 will have the *K_Event_Wait* function automatically clear the event the task is waiting for after there is a match. The task will still be notified that the event became set and that event will be cleared. Of course if the specified time out period expires then the event state will be left as it was, in the clear state.

A clear mode value of 3 instructs the *K_Event_Wait* function to perform clear mode values 1 and 2. This means the event the task is about to wait on will be forced to the clear state and when the task resumes, because the event it was waiting on happened (assuming that timeout did not occur). Clear mode values 1 and 2 are explained in the above two paragraphs.

Remember, the *K_Event_Wait* function works and manipulates only the event the calling task is requesting. Also remember, each task has a single event.

Only tasks may call this function. When called, the task will be suspended until the required match takes place or the time out period expires (if time period was non zero). When an event becomes set that the task is waiting on, the task will automatically be put back into the RESUME state, again ready to RESUME running. The value returned will indicate the event was set that the task was waiting for.

If the time period expires, the task will be placed into the RESUME state and its value returned to zero indicating a time out occurred. When a time out occurs, it allows the task to possibly take corrective action because the events did not take place within the amount of specified time.

If a task calls *K_Event_Wait* with a clear mode of 0 (zero) or 2, and the event flag are already set to the specified criteria, then the task will not be suspended.

This is an example of the *K_Event_Wait* function:

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
byte *K_Event_Wait*(byte,word16,byte); /* this is the function prototype */

#define MATCH ???

#define TIME_CNT ???

#define MODE ???

unsigned char EVENTS; /* should be local */

EVENTS = *K_Event_Wait*(MATCH,TIME_CNT,MODE);

Passed

MATCH is a 8 bit wide parameter indicating the specific event that this task would like to have set. Technically, it is only used as a single event (bit), to be upward compatible with CMX-TINY+ (which handles 8 event bits per task). Regardless of the value passed, it will be forced to 0x01 (bit 0 set).

TIME_CNT is the number of system ticks to wait for a match. If the value is 0 then the task will wait indefinitely for event match. The maximum value is 65535.

MODE is the mode in which this function will clear event bit, when the event bit is set or becomes set. The values are below.

0 = do not clear the event bits.

1 = clear the event bit according to the ones set within the MATCH parameter at BEGINNING of function.

2 = clear the event bit according to the ones set within the MATCH parameter at END of function.

3 = do both modes 1 and 2.

#define TSK2_EVENT1 0x01

```
void task2(void)
{
   unsigned char events;
   /* application code here */
```

```
   /* NOTE: could use any 1 of 3 ways to identify the events */
   events = K_Event_Wait(TSK2_EVENT1,100,2);
   /* task 2 will wait for the event bit 0 to be set.  Task 2
   will wait for 100 system ticks for this match to happen. Also
   task 2 is requesting that the clear mode command be of value
   2, which indicates that the event 0 will be automatically
   cleared when a match happens and the task RESUMES execution.
   If the time period expires, then the value returned to task 2
   will be 0, indicating that the time period expired, prior to
   the event bit 0 being set. */
   if (events == 0) /* test to see if error */
      {
      /* event 0 did NOT become set within the specified time
      period, take corrective action. */
      }
   else
      {
      /* application code here for event 0 being set */
      }
}
```

Returned

EVENTS will either contain a zero indicating the time period specified expired before any of the events the task is waiting on became set, or the specific event that were set only according to the MATCH parameter.

☞ *Remember only events that are selected by the MATCH parameter are worked within this function. The MODE parameter allows powerful synchronization as to when the task's events are cleared.*

The K_Event_Signal function

The *K_Event_Signal* function sets a specific event. This function may be called by tasks or interrupts. The caller will select which event, and the task ROM TCB index number or priority, depending upon the mode selected.

Only a single event should be set each time by the *K_Event_Signal* function.

Mode values for the K_Event_Signal function

Mode #	**Acts On**	**Task ROM_TCB index # or priority**
0 (zero)	Specific task	Task ROM_TCB index #

A mode value of 0 (zero) sets the specified event of the specified task by passing the task's ROM TCB index number. The specified task does not have to be waiting on that event. This is the fastest, for the *K_Event_Signal* function does not have to test many variables or loop. This mode can be used very effectively by tasks and interrupts.

When the *K_Event_Signal* function is called, if a task is waiting on the specified event, and the mode selects this task, then the following will occur. The task that is waiting on the event, will have its event set. This will then place the task into the READY to RESUME state, indicating that the task is able to resume its code where it left off. Also, if the task is just awakened because the event it was waiting on occurred and has a higher priority than the current RUNNING task, then rescheduling will be done after the *K_Event_Signal* function finishes, creating a task switch. Note that if an interrupt uses this call, the context switch (if applicable) will not occur until this interrupt exits and any previous lower priority interrupt (being serviced if applicable) exit as well. If a task is not waiting for the specified event and has its event set, its state will stay the same.

Remember the tasks or interrupts that call the *K_Event_Signal* function, do not have to know whether any task is waiting for an event match, or that one or more tasks are waiting on this event. The *K_Task_Wait* and *K_Task_Wake* functions can be used if only single “event” synchronization is needed. These are slightly faster.

This is an example of the *K_Event_Signal* function:

Called

Tasks and interrupts.

☞ *Interrupts can call this function. See the Processor Specific section on how to do this.*

```
#include <csfuncs.h> /* has function prototype */
byte K_Event_Signal(byte,byte,byte); /* this is the function prototype */

#define MODE ???

#define EVENT_TO_SET ???

unsigned char TASK_PRI;

unsigned char STATUS; /* should be local */

STATUS = K_Event_Signal(MODE,TASK_PRI,EVENT_TO_SET);
```

Passed

MODE is the mode in which this function will determine which tasks to work with. The values are as below.

Mode	Tasks to Work With
0	a specific task, indicated by TASK_PRI

☞ *The following parameter is not always used depending on the MODE.*

TASK_PRI is either the task ROM TCB index number or the priority in which this function will work with according to the MODE selected.

EVENT_TO_SET is an unsigned 8 bit wide variable or constant indicating the desired event bit to set.

#define Mode_0 0x00

#define TASK2_ROM_TCB_ID 2

#define TSK2_EVENT1 0x01

```
void task1(void)
{
   unsigned char status;
   /* application code here */
   status =
   K_Event_Signal(Mode_0,TASK2_ROM_TCB_ID,TSK2_EVENT1);
   /* task 1 will now set task 2's event (really bit 0), does not
   care if task 2 is waiting for event or not, if task2 is waiting
   on this event, then task 2 will automatically resume. */
}
```

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist if MODE = 0, or the MODE is out of range.

If STATUS equals K_OK, then the function performed as the MODE indicated it should.

Remember that interrupts may also call this function.

The functions *K_Event_Wait* and *K_Event_Signal*, allow a task or interrupt to signal other tasks and interrupts without the calling task or interrupt knowing that a task is waiting for its signal. For example, if task 1 is waiting for a pneumatic valve to close to continue its job responsibilities, and task 2 is waiting for task 1 to receive this signal before task 2 can go any further with its code, then task 2 would call the *K_Task_Wait* function, identifying that it will wait indefinitely, until a task or interrupt wakes it up.

Task 1 would then call the *K_Event_Wait* function indicating a match of the event flag, meaning the pneumatic valve was closed. This is sensed by an external sensor feeding its output to an interrupt. When the sensor indicated the valve had closed and generated an interrupt, the interrupt code could call *K_Event_Signal* with the proper parameters to indicate the event had happen and to set an event in task 1.

At the time of the *K_Event_Signal* call, task 1 would automatically leave the suspended state and become READY again. Task 1 then could use the *K_Task_Wake* function to notify task 2 that it had received the signal and the proper global variables were set up for task 2 to continue processing. Possibly, if the interrupt did not happen and task 1 had used the time out period of non 0, then at the end of the time period task 1 would become READY. When the task started RUNNING again it could look at the return value and if a time out had occurred by the return value being 0 (zero), it could take corrective action, such as sounding an alarm.

The K_Event_Reset function

The *K_Event_Reset* function allows a task to clear one or more specific events of a task. This means that a task can clear another task's specific event or its own. This gives you the ability to clear events, if not done so with the clear mode command within the *K_Event_Wait* function. Note, however, that tasks do not wait on events to become clear, so the respective task will not know that one of its event states was cleared unless the task is clearing its own.

This is an example of the *K_Event_Reset* function:

Called

Tasks.

```
#include <csfuncs.h> /* has function prototype */
byte K_Event_Reset(byte,byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char EVENTS_TO_CLEAR; /* the event bits to clear, or could be a #define

#define EVENTS_TO_CLEAR ??? instead of variable. */

unsigned char STATUS; /* should be local */
```

STATUS = *K_Event_Reset*(TASK1_ROM_TCB_ID,EVENTS_TO_CLEAR);

Passed

TASK1_ROM_TCB_ID is the ROM TCB index number of the task for which the event bit(s) will be cleared.

EVENTS_TO_CLEAR is a unsigned 8 bit wide variable or constant indicating the desired event bits to clear within this task. Technically, there is a single event, which always defaults to bit 0, regardless of what value is passed. This is so, to be upward compatible to CMX-TINY+.

#define TASK1_ROM_TCB_ID 1

#define TSK1_EVENT1 0x01

```
void task1(void)
{
   unsigned char status;
   status = K_Event_Reset(TASK1_ROM_TCB_ID,TSK1_EVENT1);
   /* task 1 is requesting to clear its own event bit 0. */
   .../* more task 1 code */
   K_Task_End(); /* notify CMX that the task is done */
}
```

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task ROM TCB index number does not exist.

If STATUS equals K_OK, then the event bits were cleared according to the EVENTS_TO_CLEAR parameter passed.

☞ *This function clears just the event bits within the task that is being referenced. It does not change any of the other task event bits.*

OPERATING SYSTEM FUNCTIONS

These functions are part of the CMX operating system and are described on their respective pages, also listed.

K_OS_Init (Page 28)
K_OS_Start (Page 28)
K_OS_Intrp_Entry (Page 29)
K_OS_Intrp_Exit (Page 30)

K_OS_Tick_Update (Page 30)
K_OS_Enable_Interrupts (Page 32)
K_OS_Disable_Interrupts (Page 33)

The K_OS_Init function

This function is called to initialize the CMX variables, parameters and configurable system maximums. The *K_OS_Init* function must be called before any other CMX functions are called. This is done in the user's start up code. The file containing this function should be compiled each time you change the "CPCONFIG.H" file, which declares the application's maximums.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Init*(void); /* this is the function prototype */

The K_OS_Start function

The *K_OS_Start* function is called to invoke the CMX operating system. Once this function is called, the CMX operating system takes control of the CPU and determines when tasks should run and execute. It is up to you to make sure at least one task is READY or will become READY by using the *K_Task_Start* function before calling *K_OS_Start*.

The *K_Task_Start* function call should be called within the start up code. If none of the tasks become READY, then the CMX operating system will own all the CPU time except to allow interrupts to execute.

This is an example of the *K_OS_Start* function:

Called

When you want to enter the CMX operating system.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Start*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Never returns from the CMX operating system.

K_OS_Start Example:

```
void main(void)
{
```

```
    /* define any locals within main */
    K_OS_Init(); /* initialize CMX */
    ...
    /* now the user can access any CMX function, that is allowed
    to be accessed prior to entering the CMX operating system */
    ...
    /* Set up CMX by create tasks, etc.  Start at least one task.
    */
    ...
    K_OS_Start(); /* enter into the CMX operating system */
    /* NOTE: will never return to this point */
}
```

The K_OS_Intrp_Entry function

The *K_OS_Intrp_Entry* function is used by most interrupts. The interrupt's first instruction is to call the *K_OS_Intrp_Entry* function. This informs CMX it should save the context of the CPU registers and swap in the interrupt stack when a interrupt occurs.

An interrupt does not have to call the *K_OS_Intrp_Entry* function. If it does not, then it is up to you to properly save and restore the contents of any register the interrupt will use. Also, that interrupt cannot use any CMX function calls. The interrupt's code must not call the *K_OS_Intrp_Exit* function when finished.

Please read the Processor Specific chapter below which fully describes how an interrupt must call this function.

This is an example of the *K_OS_Intrp_Entry* function:

Called

Interrupts only.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Intrp_Entry*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Nothing.

The K_OS_Intrp_Exit function

The *K_OS_Intrp_Exit* function is called by most interrupts. This function is the last instruction the interrupt's code will call. The interrupt that called the *K_OS_Intrp_Entry* should not use the instruction that signifies RETURN FROM INTERRUPT. When this function is called, CMX will automatically issue the RETURN FROM INTERRUPT instruction, informing the CPU that the interrupt has finished.

Also the CMX operating system will determine whether to restore the contexts of the task or interrupt, or to perform a rescheduling, allowing the highest priority task in the READY state to run.

Please read the Processor Specific chapter below that fully describes how an interrupt must call this function.

This is an example of the *K_OS_Intrp_Exit* function:

Called

Interrupts only.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Intrp_Exit*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Does not return.

The K_OS_Tick_Update function

This function is usually called by a timer interrupt. CMX needs one of the timer interrupts to use as a clock to perform scheduling for tasks which have used a function that invokes a time period.

The *K_OS_Tick_Update* function MUST be called by an interrupt directly or by C code from the interrupt handler. This interrupt is selectable and should cause an interrupt at a specified time period. The interrupt's frequency should be constant since all time related activities are based on this frequency. The interrupt should first call the *K_OS_Intrp_Entry* function, then call the *K_OS_Tick_Update* function. The *K_OS_Tick_Update* function will decrement a counter that has been pre-loaded with the number of times the interrupt must call this function, before servicing the task timers. The variable RTC_SCALE defaults to a vaule of one (1), though the user may change this value by directly writing a value to this variable, which is an unsigned char.

When the count specified by the RTC_SCALE reaches zero, then the counter will be reloaded with the RTC_SCALE value, and any task timers that need servicing will have their respective timers decremented. If a task's timer reaches zero, the task will be taken out of the SUSPENDED state and put in the READY TO RESUME state. When the interrupt leaves the *K_OS_Intrp_Exit* function, the scheduler will determine whether to reschedule or to resume execution of the current running task. (See the Scheduler chapter for a more detailed description on how the scheduler works).

This is an example of the *K_OS_Tick_Update* function:

Called

By the interrupt handler directly or by the interrupt handler calling C code, which then calls this function.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Tick_Update*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Nothing.

K_OS_Tick_Update Example:

```
void __attribute__((__interrupt__,)) _T1interrupt (void)
{
    /* The following MUST be the FIRST C statement. */
    K_OS_Intrp_Entry(); /* Call the CMX interrupt prologue function. */
    K_OS_Tick_Update(); /* Call the CMX system tick function. */

    /* MORE user code would go here, if need be, such as clearing timer flag. */
    IFS0bits.T1IF = 0; /* reset timer 1 flag */

    /* The following MUST be the LAST C statement. */
    K_OS_Intrp_Exit(); /* Call the CMX interrupt epilogue function. */
}
```

The K_OS_Enable_Interrupts function

This function GLOBALLY enables interrupts. It is used in conjunction with the *K_OS_Disable_Interrupts* function. All non-maskable interrupts that were GLOBALLY disabled, will be enabled. If any interrupt is pending, then the interrupt will now be recognized and processed according to the CPU interrupt hardware mechanism and priority scheme.

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Enable_Interrupts*(void); /* this is the function prototype */

Passed

Nothing is passed.

Returned

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect on interrupts when they are GLOBALLY enabled.

K_OS_Enable_Interrupts example:

```
void task1(void)
{
   unsigned char status; /* local */
   .../* application code here */
   K_OS_Disable_Interrupts(); /* do critical region of code
   items. */
   ...
   K_OS_Enable_Interrupts(); /* re-enable interrupts. */
   .../* application code here */
}
```

Comments

This function should be called as soon as possible after the *K_OS_Disable_Interrupts* function is called. Some C vendor compilers have special instructions that allow the interrupts to be GLOBALLY enabled and the code placed inline. This executes faster than calling the equivalent CMX function.

The K_OS_Disable_Interrupts function

This function GLOBALLY disables the interrupts. Any non-maskable interrupt will not be immediately recognized. If the interrupt sets a latch, then this will not be prevented. This uses the particular CPU instruction that masks out all non-maskable interrupts from being acknowledged and processed.

Called

Tasks.

#include <csfuncs.h> /* has function prototype */
void *K_OS_Disable_Interrupts*(void); /* this is the function prototype */

Passed

Nothing is passed.

Returned

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect it has on interrupts when the interrupts are GLOBALLY disabled.

K_OS_Disable_Interrupts example:

```
void task1(void)
{
   unsigned char status; /* local */
   .../* application code here */
   K_OS_Disable_Interrupts(); /* do critical region of code
   stuff. */
   ...
   K_OS_Enable_Interrupts(); /* re-enable interrupts. */
   .../* application code here */
}
```

Comments

Use this function sparingly or not at all if possible. ALL maskable interrupts will not be recognized until the interrupts are re-enabled. This will also add latency time to the interrupt processing its code. If used, the user must remember to re-enable interrupts using the *K_OS_Enable_Interrupts* function.

Some C vendor compilers have special instructions that will allow the interrupts to be GLOBALLY disabled and the code placed inline. This executes faster than calling the equivalent CMX function.

PROCESSOR SPECIFIC CHAPTER

CMX needs a constant frequency interrupt, which is normally a timer. This interrupt should CALL the CMX *K_OS_Tick_Update* function, which is used for all timed based operations that are related to CMX. This includes the task timers. It is up to the user to determine which timer (or other source of a constant periodic frequency) will be used and what the frequency should be set for.

dsPic Interrupts and Interfacing to CMX

A preempted operating system must determine when a higher priority task should run, therefore preempting the current running task. Remember that an interrupt may cause a higher priority task to be ready to run, therefore causing the scheduler to immediately (after the interrupts code has finished) save the current executing task (the one prior to the interrupt) contexts and let the higher priority task start execution immediately. The only way to achieve this, is by having interrupts increment and decrement a counter to let the interrupt handler know when the 1'st interrupt has finished it's code, because nested interrupt are allowed. Note that some operating systems will have the "Operating interrupt handler" increment and decrement this count, while others will have the interrupt code perform this. The CMX interrupt handler does this automatically for you.

Since the dsPic has quite a number of registers, CMX does NOT save all the registers when an interrupt occurs. The compiler will save the following when the interrupt keyword is used: RCOUNT, W0-W7. The RTOS will save W8-W14. It is up to the user to ensure that either they or the compiler saves and restores any additional registers that may be used within the interrupt.

The following describes how the user should code the interrupt, so as to be compatible with CMX. CMX will describe interrupt routines that will allow other interrupts to happen, disable other interrupts from happening and having that interrupt interface to either the CMX *K_Event_Signal* and CMX *K_OS_Tick_Update* or not.

One additional note is that the CMX *K_OS_Intrp_Entry* assembly code returns to the interrupt handler with STATUS REGISTER (SR) having the interrupt level field (IPL) the same value that it previously was to calling *K_OS_Intrp_Entry* function. The user is then free to modify this level if they so choose to. Also the CMX *K_OS_Intrp_Exit* assembly code will automatically disable interrupts for a short period of time.

The user should carefully read the dsPic manual, that fully describes the interrupts and how they work within this processor. This will let the user better understand this chapter and also will allow the user to understand the *K_OS_Intrp_Entry* and *K_OS_Intrp_Exit* routines.

Interrupts and Coding

In most cases, interrupt handlers must be written in C, at least the prologue and epilogue portion as follows:

```
void __attribute__((__interrupt__,)) [VECTOR_LOCATION] (void)
{
    /* The following MUST be the FIRST C statement. */
    K_OS_Intrp_Entry(); /* Call the CMX interrupt prologue function. */

    /* Now the user code would go here. *.

    /* The following MUST be the LAST C statement. */
    K_OS_Intrp_Exit(); /* Call the CMX interrupt epilogue function. */
}
```

For those of you writing only in assembly or want to code the interrupt handler in assembly, you MUST follow the following exactly, since this is what the RTOS expects it in this order.

```
.align4
.global__T1Interrupt; export
__T1Interrupt:
; This is the Prologue, must be exactly as this
push  _RCOUNT
mov.d w0,[w15++]
mov.d w2,[w15++]
mov.d w4,[w15++]
mov.d w6,[w15++]
K_OS_Intrp_Entry

;Now user can do what they want to within their handler code

; This is the Epilogue, must be exactly as this
K_OS_Intrp_Exit
mov.d [--w15],w6
mov.d [--w15],w4
mov.d [--w15],w2
mov.d [--w15],w0
pop   _RCOUNT
retfie
```

The following shows an interrupt that will call the CMX *K_OS_Tick_Update* function, which is the 'SYSTEM TICK'.

```
void __attribute__((__interrupt__,)) _T1interrupt (void)
{
    /* The following MUST be the FIRST C statement. */
    K_OS_Intrp_Entry(); /* Call the CMX interrupt prologue function. */
    K_OS_Tick_Update(); /* Call the CMX system tick function. */

    /* MORE user code would go here, if need be, such as clearing timer flag. */
    IFS0bits.T1IF = 0; /* reset timer 1 flag */

    /* The following MUST be the LAST C statement. */
    K_OS_Intrp_Exit(); /* Call the CMX interrupt epilogue function. */
}
```

There is 1 scenario where an interrupt may not need to call the CMX *K_OS_Intrp_Entry* function. This is an interrupt that will not call ANY CMX functions (*K_OS_Intrp_Entry*, *K_OS_Intrp_Exit* or *K_Event_Signal*) and will IPL level greater then any interrupt that would call any of the CMX functions allowable by an interrupt (*K_OS_Intrp_Entry*, *K_OS_Intrp_Exit* or *K_Event_Signal*) .

```
void __attribute__((__interrupt__,)) [VECTOR_LOCATION] (void)
{
    /* This interrupt will NOT call K_OS_Intrp_Entry, K_OS_Intrp_Exit or
    K_Event_Signal. It must NOT lower the IPL level within the STATUS SR register
    and it can NOT call any CMX functions. */
    /* Now the user code would go here. *.

}
```

interrupt Using The K_Event_Signal function

The user should be sure to read the preceding pages on how to interface interrupts to the CMX *K_Event_Signal* or *K_OS_Tick_Update* function, which explains the prologue and epilogue requirements. This is shown in the following examples and it is ASSUMED it will be done.

CMX highly recommends that the user code their *K_Event_Signal* function in C, thus allowing the compiler to generate the code, so as the *K_Event_Signal* function receives the parameters properly. An example is below:

```
void __attribute__((__interrupt__,)) [VECTOR_LOCATION] (void)
{
```

```
    /* The following MUST be the FIRST C statement. */
    K_OS_Intrp_Entry(); /* Call the CMX interrupt prologue function. */

    K_Event_Signal(0,TASK1_RAM_POS,0x04); /* signal event 0x04 (bit 2) to task1,
    using mode zero. */

    /* The following MUST be the LAST C statement. */
    K_OS_Intrp_Exit(); /* Call the CMX interrupt epilogue function. */
}
```

CONFIGURING THE RTOS

The user MUST do the following within their code that has main.

The variable name 'int_storage' MUST be used, since the RTOS will look for this name.

```
#define C_INTERRUPT_SIZE 128 /* Size of interrupt stack. This is
used by interrupts, The CMX timer task and the scheduler. */

struct {
word16 interrupt_bytes[C_INTERRUPT_SIZE / 2];
} int_storage;
```

The C_INTERRUPT_SIZE is the size of the stack used for first and subsequent interrupts acknowledged by the CPU. This stack is also used by the CMX scheduler and the CMX timer task when its contents are being executed. You need enough stack space for the deepest nested interrupts stack use. The way to calculate this value is the maximum number of nested interrupts that could occur multiplied by the number and size of the registers for the particular processor. Also add the number of bytes that each interrupt could push on the stack. This is a tough calculation and CMX highly recommends you double the calculated size initially. CMX feels a minimum number is 128 bytes is a good starting point.

The user MUST define the ROM TCB array. The name for the array MUST be 'task_rom_list'

```
unsigned short task_stack[200];        /* global array for task stacks.  */

ROM_TCB task_rom_list[]= {

    /* The first entry MUST be declared as follows, used by the RTOS.  */
    main, 0, 0,
    task1, &task_stack[0], 5,
    task2, &task_stack[50], 3,
    task3, &task_stack[100], 6,
    task4, &task_stack[150], 2
    };
```

The first task within the ROM_TCB must be the the proram's *main function*. The *main* function uses the startup stack, so it is not necessary to specify a stack size for it. The main function is really NOT a task, but is needed here. The stack for each task is assumed to grow UPWARDS (From low to high memory). The size of a particular task stack must be large enough to hold the following: the registers, the number of return addresses (generated by calls to functions), the number of local variables within the functions called and the number of arguments placed on the stack by a calling function.

You MUST be careful that the size of the task stack is large enough. Failure to do so will result in system crashes. We recommend that the initial stack size be doubled over the estimated size until testing shows the size can be reduced.

As you can see, there are 4 user tasks in the ROM TCB. The *main function MUST* be the first task listed.

The RTC_SCALE counts the number of times the CMX function *K_OS_Tick_Update* is called by an interrupt before the *K_OS_Tick_Update* function sets the timer flag, if there are task timers and/or cyclic timers that need servicing. The timer flag notifies the scheduler that it should now save the contexts of the current RUNNING task and let the CMX timer task execute.

The interrupt that calls the CMX *K_OS_Tick_Update* function should be an interrupt you can program to occur consistently at the time interval for this interrupt. Most CPU's have an onboard timer that can be programmed to cause an interrupt at a specified time.

To get a multiple of this interrupt's time before the timer flag is set, determine what multiple you want. The RTC_SCALE is a value from one to 255. For example, say a particular CPU has the ability to program one of its timers to cause an interrupt every 10 milliseconds accurately. Now you want to base all time related activities at every 30 millisecond interval. The value set for the RTC_SCALE would be three. The higher the system tick frequency (you determine the scheduling frequency), the more the CMX scheduler and CMX timer task will be invoked, and the less time spent on tasks running.

As another example, say it takes on average 100 microseconds to perform a task switch. (NOTE: some processors take a lot less time, some more, this depends on the number of tasks in the application code.) Also you have specified a scheduling frequency of 5 milliseconds. Therefore the CPU's time will be the following: 2% on scheduling and 98% on letting tasks RUN. The CMX timer task will also take away from the tasks execution time. The amount of time depends on the number of tasks waiting on at least time (they may be waiting on another entity also, such as an event) and the number of cyclic timers that are currently running and whether or not they are to execute their respective *K_Event_Signal* function. Also, keep in mind that interrupts will also steal the CPU time from tasks, the scheduler, CMX timer task and possibly other interrupts (if nesting of interrupts is allowed).

The variable RTC_SCALE defaults to a value of one (1), though the user may change this value by directly writing a value to this variable, which is an unsigned char. as an example.

```
RTC_SCALE = 5;
```

STACKS IN GENERAL

This chapter will discuss both task and interrupt stacks. Since all C compilers place arguments (parameters) on the stack, functions (tasks are functions) use the stack to allocate local data space and save the return address, it is very important for you to understand this chapter.

On most microprocessors and microcomputers the stack resides in external memory, though some processors have an internal stack. There are slight differences among processors. The way processors work with the stack are described later in the Processor Specific Information chapter.

All processors have a stack pointer register (some are dedicated, others use a specific register according to the C compiler manufacturer). Also some C manufacturers create a frame pointer to access the locals created within a function and also to access the parameters passed to a function. Again the frame pointer is a dedicated register, or a register chosen by the C compiler manufacturer.

CMX lets you configure the size of external memory to set aside for all task stacks and the size for the interrupt stack. Each task will have its own stack. This allows CMX to save a task's context very fast. When a task's contexts are saved, just the registers have to be pushed on to the stack. Then the stack pointer gets loaded with another memory address. (The memory address swapped in will be the interrupt stack or another task's stack.)

It is up to you to ensure each task stack and the interrupt stack are large enough in size. The task's stack must be large enough to hold that particular task's local data area, all return addresses that are pushed on to the stack as each function is called and all called functions' local data areas.

Each of the task's stacks must be able to hold all of the CPU's REGISTERS in case of an interrupt. If the CPU has a "non-maskable interrupt", then the task's stack must be capable of holding the additional number of bytes. This is so the CPU's REGISTERS will be saved when the non-maskable interrupt occurs.

We will show how to calculate the stack size needed and provide two examples. It is recommended you double the estimated size of all task's stacks and the interrupt stack, at least initially. As you become more proficient and code testing indicates the stack size may be decreased, then the stack sizes may be reduced.

Unpredictable results can occur when either a task's stack or the interrupt stack pushes into memory that is not a part of this stack's memory.

A warning about the examples. All C compilers are not exactly the same in the way a function is called (some use a stack frame, some do not) CMX highly recommends you study the C compiler manual that describes stack usage, locals and parameter passing, and the code (assembly source) produced by the C compiler to better understand the particular C compiler stack usage.

These are examples only. CMX does not guarantee this is exactly the true number of bytes needed for the tasks and interrupt stack. The examples below assume ints are (2) two bytes long. On some processors ints will be (4) four bytes long.

Example 1: (a lot of locals, no nested function calls)

```
void Task1(void) /* tasks may not receive nor return parameters)
{
   /* note that some C compilers create a stack frame which will
   use some of this task stack space to save the stack frame
   pointer and possibly other registers */
   int a; /* 2 bytes of local space */
   char b; /* 1 byte of local space *
   char *ptr; /* 2 bytes of local, for processors that address
   up to 65535 bytes and no bank switching.  Also NO alignment
   boundaries. */
```

☞ *Some processors need integers, pointers, etc. to be aligned properly to the processor alignment boundary. In some cases this is at an EVEN memory address, other times it at an ODD memory address. If the local is aligned incorrectly, according to the processors alignment rule, then an additional byte will be used in the local space. Be aware of this.*

```
   func_A(char,int); /* func_A is a function that will receive a
   character and an integer.  3 bytes worth of stack space PLUS
   2 bytes for the return address */
   func_B(int,int); /* func_B is a function that will receive an
   integer and an integer.  4 bytes worth of stack space PLUS 2
   bytes for the return address */

}
void func_A(char,int)
{
   char a; /* 1 byte of local really used from task 1 stack area
   */
   int b; /* 2 bytes of local, really used from task 1 stack area
   */
   ..../* func_A code */
}

void func_B(int,int)
{
   int a; /* 2 bytes of local really used from task 1 stack area
   */
```

```
   int b; /* 2 bytes of local, really used from task 1 stack area
   */
   ..../* func_B code */
}
```

In the above example, the number of bytes used from task 1 stack area would be five bytes for task 1 locals, four bytes for func_B arguments, two bytes for func_B return address and four bytes for func_B locals. Make note that func_A is not calculated because func_B uses more memory than func_A.

This calculation does not include saving the stack frame pointer, if one is used, nor ALIGNMENT if the CPU requires it.

Example 2: (a lot of locals, some nested function calls)

```
void Task1(void) /* tasks may not receive nor return parameters)
{
   /* note that some C compilers create a stack frame which will
   use some of this task stack space to save the stack frame
   pointer and possibly other registers */
   int a; /* 2 bytes of local space */
   char b; /* 1 byte of local space *
   char *ptr; /* 2 bytes of local, for processors that address
   up to 65535 bytes and no bank switching.  Also NO alignment
   boundaries. */
```

☞ *Some processors need integers, pointers, etc. to be aligned properly to the processor alignment boundary. In some cases this is at an EVEN memory address, other times it at an ODD memory address. If the local is aligned incorrectly, according to the processors alignment rule, then an additional byte will be used in the local space. Be aware of this.*

```
   func_A(char,int); /* func_A is a function that will receive a
   character and an integer.  3 bytes worth of stack space PLUS
   2 bytes for the return address */

   func_B(int,int); /* func_B is a function that will receive an
   integer and an integer.  4 bytes worth of stack space PLUS 2
   bytes for the return address */
}

void func_A(char,int)
{
   char a; /* 1 byte of local really used from task 1 stack area
   */
   int b; /* 2 bytes of local, really used from task 1 stack area
   */
   ..../* func_A code */
```

```
   func_C(int,int); /* nested function call to func_C.  Func_C
   is a function that will receive 2 integers.  4 bytes worth of
   task 1 stack space PLUS 2 bytes for the return address */
}

void func_B(int,int)
{
   int a; /* 2 bytes of local really used from task 1 stack area
   */
   int b; /* 2 bytes of local, really used from task 1 stack area
   */
   ..../* func_B code */
   }
   void func_C(int,int)
   {
   int a; /* 2 bytes of local really used from task 1 stack area
   */
   int b; /* 2 bytes of local, really used from task 1 stack area
   */
   ..../* func_C code */
}
```

In the above example, the number of bytes used from task 1 stack area would be five bytes for task 1 locals, three bytes for func_A arguments, two bytes for func_A return address, three bytes for func_A locals, four bytes for func_C arguments, two bytes for func_C return address, four bytes for func_C locals. Make note that func_B is not calculated because func_A that calls func_C uses more memory than func_B.

This calculation does not include saving the stack frame pointer, if one is used, nor ALIGNMENT if the CPU requires it.

As you can see, the task 1 stack gets used for task 1 locals and all functions that are called by task 1. This includes the functions' locals and the return address for all the functions that task 1 will call.

CMX suggests you enter a pattern into all the tasks' stack area, like the value AA hex, and then run the application code. Then you may view each particular task's stack area and see how much of the task's stack has really been used. If the stack pushes down pass its allocated memory, then serious negative results will occur, with the system crashing and corrupting memory.

THE CMX SCHEDULER CHAPTER

This chapter will explain in detail how the CMX scheduler works. The CMX scheduler is based on true preemption. This means that tasks and interrupts can cause an immediate task switch if a higher priority task becomes able to run as a result of a CMX function. Cooperative scheduling is also possible, so a task can let the next task (lower or same priority) run if so desired. This gives you complete control on how tasks will execute.

In CMX, a task will be in one of the following states:

IDLE state

A task that has been created with the *K_Task_Create* function, but not started by the *K_Task_Start* function is in the IDLE state. A task that has completed its code and called the *K_Task_End* function is placed into the IDLE state if there are no outstanding triggers in its control block. A task that is in its IDLE state will not run.

READY state

The READY state informs the scheduler that the task is ready to run, but NOT running. This allows the scheduler to determine what task to run when a scheduling takes place according to the task's priority in relation to the other tasks' priorities.

RUN state

The task that is executing is in the RUN state and owns the CPU time. Only one task may be in the RUN state at any one time.

WAIT (suspended) state

A task that suspended itself by a CMX function call is in the WAIT (suspended) state. There are many function calls that will suspend a task. The WAIT state consists of a task that is waiting on one or more of the following: time, events, flags, messages, on a reply, etc.

RESUME state

The RESUME state is treated the same as the READY state. The only difference is that it informs the scheduler that the task had been started, yet not finished, with its code. This means that a higher priority task has preempted and forced the original task that was running to become ready to RESUME, or that the task had suspended itself by a function call and now has removed itself out of the WAIT (suspended) state into the ready to RESUME state.

KILL state

A task that is KILLED (removed) has its state put into the KILL state, indicating this task no longer exists.

QUICK REFERENCE

The following pages are a quick reference to all the CMX functions. The functions are listed within their respective manager in a alphabetical order so you can easily look up the desired function for reference. These are intended to assist the user in syntax and use of the function.

[Before entering RTOS] means the call may be used prior to entering the CMX operating system. [Tasks] mean a task may use this CMX call. [Interrupts] means interrupts may call this CMX function.

☞ *The K_OS_Init function must be called before any CMX function call may be used.*

Each CMX call also lists what CMX parameters or arguments are needed and passed, and a description indicating what should be loaded into these parameters or arguments. The CMX call will show the return STATUS parameter values most CMX functions return. If STATUS is returned, it is of unsigned character size.

An example is shown using the parameters or arguments needed for this CMX call, with the CMX call. For a detailed description of each CMX function, please refer to the appropriate function manager chapter at the beginning of this document.

CMX RETURN STATUS BYTE VALUES

Symbol	Hex	Value	Explanation
K_OK	00	Good	CMX call was successful
K_TIMEOUT	01	Warning / Error	Time out occurred
K_NOT_WAITING	02	Error	Task not waiting for wake request
K_ERROR	FF	Error	General error, CMX call unsuccessful

Some functions, such as *K_Mesg_Get*, may return a NULL pointer if there is no message available. This indicates a possible warning/error to the caller, for NO message was retrieved. In some cases a return value of zero indicates the CMX function is telling the caller the item it wanted was not there or a time out occurred.

EVENT MANAGER FUNCTION

K_Event_Reset

Purpose: This function clears (resets) one or more specific events within a task. This does not change a task state, nor does it specify which events were set or not. In most cases, this function is not needed. The *K_Event_Wait* function automatically clears the events of a task, if so programmed.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
byte *K_Event_Reset*(byte,byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char EVENTS_TO_CLEAR; /* the event bits to clear, or could be a #define

#define EVENTS_TO_CLEAR ??? instead of variable. */

unsigned char STATUS; /* should be local */

STATUS = *K_Event_Reset*(TASK1_ROM_TCB_ID,EVENTS_TO_CLEAR);

Passed

TASK1_ROM_TCB_ID is the ROM TCB index number of the task for which the event bit(s) will be cleared.

EVENTS_TO_CLEAR is a unsigned 8 bit wide variable or constant indicating the desired event bits to clear within this task. Technically, there is a single event, which always defaults to bit 0, regardless of what value is passed. This is so, to be upward compatible to CMX-TINY+.

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist.

If STATUS equals K_OK, then the event bits were cleared according to the EVENTS_TO_CLEAR parameter passed.

☞ *This function clears just the event bits within the task that is being referenced. It does not change any of the other task event bits.*

K_Event_Reset example:

#include <cpfuncs.h> /* has function prototype */

#define TASK1_ROM_TCB_ID 1

#define TSK1_EVENT1 0x01

```
void task1(void)
{
   unsigned char status;
   status = K_Event_Reset(TASK1_ROM_TCB_ID,TSK1_EVENT1);
   /* task 1 is requesting to clear its own event bit 0. */
   if (status != K_OK)
      {
      /* maybe take corrective action. */
      }
   ...
}
```

EVENT MANAGER FUNCTION

K_Event_Signal

Purpose: The *K_Event_Signal* function sets a specific event. This can be done in a variety of modes. This function may be called by tasks or interrupts. The caller will select which event, the mode of event set it wants, and the task ROM TCB index number or priority, depending upon the mode selected.

Called

Tasks and interrupts.

☞ *Interrupts can call this function. See the Processor Specific section on how to do this.*

```
#include <cpfuncs.h> /* has function prototype */
byte K_Event_Signal(byte,byte,byte); /* this is the function prototype */

#define MODE ???

unsigned char TASK_PRI;

#define EVENT_TO_SET ???

unsigned char STATUS; /* should be local */

STATUS = K_Event_Signal(MODE,TASK_PRI,EVENT_TO_SET);
```

Passed

MODE is the mode in which this function will determine which tasks to work with. The values are below.

Mode #	Acts On	Task ROM_TCB index # or priority
0 (zero)	Specific task	Task ROM_TCB index #

TASK_PRI is either the task ROM TCB index number or the priority in which this function will work with according to the MODE selected.

☞ *This parameter is not always used depending on the MODE.*

EVENT_TO_SET is an unsigned 8 bit wide variable or constant indicating the desired event bit to set.

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task does not exist if MODE = 0, or the MODE is out of range.

If STATUS equals K_OK, then the function performed as the MODE indicated it should.

Remember that interrupts may also call this function.

K_Event_Signal example:

#define TASK2_ROM_TCB_ID 2

#define EVENT1 0x01 /* define event bit 0 */

```
void task1(void)
{
   unsigned char status; /* local */
   ...
   status = K_Event_Signal(0,TASK2_ROM_TCB_ID,EVENT1);
   /* task 1 will tell the K_Event_Signal function to set event
   bit 0 of only task 2, whether task 2 is waiting or not for
   this event.  If task 2 is waiting, then task 2 will
   automatically resume. */
   if (status != K_OK)
      {
      /* take corrective action. */
      }
   ...
}
```

Comments

This function is very flexible and powerful. As you can see, it can be used by tasks and interrupts.

EVENT MANAGER FUNCTION

K_Event_Wait

Purpose: This function allows a task to wait for specific event with a specified time out if desired. The task will also specify the mode that indicates when to automatically clear the event bit that match. The task will be suspended until either the number of system ticks has expired or the event bit it is waiting for is set or becomes set. A time period of zero indicates the task will wait indefinitely for an event match. If there is a match as specified when the task calls this function, the task will not be suspended and will immediately be returned to.

Called

Tasks.

```
#include <cpfuncs.h> /* has function prototype */
byte K_Event_Wait(byte,word16,byte); /* this is the function prototype */

#define MATCH ???

#define TIME_CNT ???

#define MODE ???

unsigned char EVENTS; /* should be local */

EVENTS = K_Event_Wait(MATCH,TIME_CNT,MODE);
```

Passed

MATCH is a 8 bit wide parameter indicating the specific event that this task would like to have set. Technically, it is only used as a single event (bit), to be upward compatible with CMX-TINY+ (which handles 8 event bits per task). Regardless of the value passed, it will be forced to 0x01 (bit 0 set).

TIME_CNT is the number of system ticks to wait for a match. If the value is 0 then the task will wait indefinitely for event match.

MODE is the mode in which this function will clear event bits, when they are set or become set. The values are below.

0 = do not clear the event bit.

1 = clear the event bit according to the ones set within the MATCH parameter at BEGINNING of function.

2 = clear the event bit according to the ones set within the MATCH parameter at END of function.

3 = do both modes 1 and 2.

Returned

EVENTS will either contain a zero indicating the time period specified expired before the event the task is waiting on became set or were set according to the MATCH parameter.

☞ *Remember only a single event is selected by the MATCH parameter are worked within this function. The MODE parameter allows powerful synchronization as to when the task's events are cleared.*

K_Event_Wait example:

#define TSK2_EVENT1 0x01

```
void task2(void)
{
   unsigned char events;
   /* application code here */
   /* NOTE: could use any 1 of 3 ways to identify the events */
   events = K_Event_Wait(TSK2_EVENT1,100,2);
   /* task 2 will wait for the event bit 0 to be set.  Task 2
   will wait for 100 system ticks for this match to happen. Also
   task 2 is requesting that the clear mode command be of value
   2, which indicates that the event 0 will be automatically
   cleared when a match happens and the task RESUMES execution.
   If the time period expires, then the value returned to task 2
   will be 0, indicating that the time period expired, prior to
   the event bit 0 being set. */
   if (events == 0) /* test to see if error */
      {
      /* event 0 did NOT become set within the specified time
      period, take corrective action. */
      }
   else
      {
      /* application code here for event 0 being set */
      }
}
```

OPERATING SYSTEM FUNCTION

K_OS_Disable_Interrupts

Purpose: This function GLOBALLY disables the interrupts. Any non-maskable interrupt will not be immediately recognized. If the interrupt sets a latch, then this will not be prevented. This uses the particular CPU instruction that masks out all non-maskable interrupts from being acknowledged and processed.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
void *K_OS_Disable_Interrupts*(void); /* this is the function prototype */

Passed

Nothing is passed.

Returned

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect it has on interrupts when the interrupts are GLOBALLY disabled.

K_OS_Disable_Interrupts example:

```
void task1(void)
{
   unsigned char status; /* local */
   .../* application code here */
   K_OS_Disable_Interrupts(); /* do critical region of code
   stuff. */
   ...
   K_OS_Enable_Interrupts(); /* re-enable interrupts. */
   .../* application code here */
}
```

Comments

Use this function sparingly or not at all if possible. ALL maskable interrupts will not be recognized until the interrupts are re-enabled. This will also add latency time to the interrupt processing its code. If used, the user must remember to re-enable interrupts using the *K_OS_Enable_Interrupts* function.

Some C vendor compilers have special instructions that will allow the interrupts to be GLOBALLY disabled and the code placed inline. This executes faster than calling the equivalent CMX function.

OPERATING SYSTEM FUNCTION

K_OS_Enable_Interrupts

Purpose: This function GLOBALLY enables interrupts. It is used in conjunction with the *K_OS_Disable_Interrupts* function. All non-maskable interrupts that were GLOBALLY disabled, will be enabled. If any interrupt is pending, then the interrupt will now be recognized and processed according to the CPU interrupt hardware mechanism and priority scheme.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
void *K_OS_Enable_Interrupts*(void); /* this is the function prototype */

Passed

Nothing is passed.

Returned

Nothing is returned.

Read the particular CPU manual that further describes the action and the effect on interrupts when they are GLOBALLY enabled.

K_OS_Enable_Interrupts example:

```
void task1(void)
{
   unsigned char status; /* local */
   .../* application code here */
   K_OS_Disable_Interrupts(); /* do critical region of code
   items. */
   ...
   K_OS_Enable_Interrupts(); /* re-enable interrupts. */
   .../* application code here */
}
```

Comments

This function should be called as soon as possible after the *K_OS_Disable_Interrupts* function is called. Some C vendor compilers have special instructions that allow the interrupts to be GLOBALLY enabled and the code placed inline. This executes faster than calling the equivalent CMX function.

OPERATING SYSTEM FUNCTION

K_OS_Init

Purpose: This function is called to set up all the memory needed for CMX and initialize the CMX variables, parameters and configurable system maximums. The *K_OS_Init* function must be called before any other CMX functions are called. This is done in the start up code. The file containing the function *K_OS_Init* should be compiled each time you change the "CPCONFIG.H" file, which declares the application's maximums.

Called

Before using any other CMX function calls.

#include <cpfuncs.h> /* has function prototype */
void *K_OS_Init*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Nothing.

☞ *Remember this function must be called before any other CMX function is called. If not, then disastrous results will occur.*

K_OS_Init Example:

```
void main(void)
{
   /* define any locals within main */
   K_OS_Init(); /* initialize CMX */
   ...
   /* now the user can access any CMX function, that is allowed
   to be accessed prior to entering the CMX operating system */
   ...
}
```

OPERATING SYSTEM FUNCTION

K_OS_Intrp_Entry

Purpose: The *K_OS_Intrp_Entry* function is used by most interrupts. The interrupt's first instruction is to call the *K_OS_Intrp_Entry* function. This informs CMX it should save the context of the CPU registers and swap in the interrupt stack when a interrupt occurs. See the chapter on Processor Specific Information for details on the particular CPU you are working with.

Because some C compilers generate code (such as setting up the frame pointer for a function), the interrupt handler, for most processors, should be written in assembly language. The interrupt handler follows the guidelines for the particular CPU calling the *K_OS_Intrp_Entry* routine. After this is done, the interrupt handler may call the interrupt processing code written in C, or just continue with the assembly language interrupt processing code.

It is recommended to write all interrupt code in assembly, since greater speed is achieved. If the interrupt does not call this function, then the interrupt must save and restore all registers used, and this interrupt may not use any CMX calls. Please read the Processor Specific chapter on interrupts and the additional interrupt notes supplied for the CPU you are working with.

OPERATING SYSTEM FUNCTION

K_OS_Intrp_Exit

Purpose: Any interrupt that used the *K_OS_Intrp_Entry* call to have CMX save the current task context and switch in the interrupt stack, must use this function when the interrupt has finished its code. This takes the place of the normal CPU's return from interrupt instruction. This function is the last instruction of the interrupt's code.

Please read the Processor Specific chapter on interrupts and the additional interrupt notes supplied for the CPU you are working with.

OPERATING SYSTEM FUNCTION

K_OS_Start

Purpose: The *K_OS_Start* function is called to invoke the CMX operating system. Once this function is called, the CMX operating system takes control of the CPU and determines when tasks should run and execute. It is up to you to make sure at least one task is READY or will become READY by using the *K_Task_Start* function before calling *K_OS_Start*. Once you enter the CMX operating system, there is no way to exit the CMX operating system.

Called

When you want to enter the CMX operating system.

#include <cpfuncs.h> /* has function prototype */
void *K_OS_Start*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Never returns from the CMX operating system.

K_OS_Start Example:

```
void main(void)
{
   /* define any locals within main */
   K_OS_Init(); /* initialize CMX */
   ...
   /* now the user can access any CMX function, that is allowed
   to be accessed prior to entering the CMX operating system */
   ...
   /* Set up CMX by create tasks, etc. Start at least one task. */
   ...
   K_OS_Start(); /* enter into the CMX operating system */
   /* NOTE: will never return to this point */
}
```

OPERATING SYSTEM FUNCTION

K_OS_Tick_Update

Purpose: This function is usually called by a timer interrupt. CMX needs one of the timer interrupts to use as a clock to perform scheduling for tasks which have used a function that invokes a time period.

The *K_OS_Tick_Update* function MUST be called by an interrupt directly or by C code from the interrupt handler. This interrupt is selectable and should cause an interrupt at a specified time period. The interrupt's frequency should be constant since all time related activities are based on this frequency. The interrupt should first call the *K_OS_Intrp_Entry* function, then call the *K_OS_Tick_Update* function. The *K_OS_Tick_Update* function will decrement a counter that has been pre-loaded with the number of times the interrupt must call this function, before servicing the timers. This counter is loaded with a value you select by the variable RTC_SCALE, as described in the RTOS Configuration the RTOS chapter.

When the count specified by the RTC_SCALE reaches zero, then the counter will be reloaded with the RTC_SCALE value, and any task timers that need servicing will have their respective timers decremented. If a task's timer reaches zero, the task will be taken out of the SUSPENDED state and put in the READY TO RESUME state. When the interrupt leaves the *K_OS_Intrp_Exit* function, the scheduler will determine whether to reschedule or to resume execution of the current running task.

Called

By the interrupt handler directly or by the interrupt handler calling C code, which then calls this function.

#include <cpfuncs.h> /* has function prototype */
void *K_OS_Tick_Update*(void); /* this is the function prototype */

Passed

Nothing.

Returned

Nothing.

K_OS_Tick_Update Example:

```
void __attribute__((__interrupt__,)) _T1interrupt (void)
{
    /* The following MUST be the FIRST C statement. */
    K_OS_Intrp_Entry(); /* Call the CMX interrupt prologue function. */
    K_OS_Tick_Update(); /* Call the CMX system tick function. */
```

```
    /* MORE user code would go here, if need be, such as clearing timer flag. */
    IFS0bits.T1IF = 0; /* reset timer 1 flag */

    /* The following MUST be the LAST C statement. */
    K_OS_Intrp_Exit(); /* Call the CMX interrupt epilogue function. */
}
```

TASK MANAGER FUNCTION

K_Task_Coop_Sched

Purpose: This function performs cooperative rescheduling. This allows a task the ability to let a task of the same or lower priority become the current RUNNING task. Usually there is little need for this because of the preemptive nature of the CMX operating system. However, there may be times you would want a lower priority task to execute before this task would normally do so.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
void *K_Task_Coop_Sched*(void); /* this is the function prototype */

Passed

Nothing is passed. Only tasks can call this function.

Returned

No status is returned.

K_Task_Coop_Sched example:

void task2(void); /* prototype task2 */

```
void task2(void)
  {
  /* task's application code ... */
  K_Task_Coop_Sched();
  /* task 2 has decided to release control to the next same
  priority or lower priority task able to run. */
  ...
 }
```

TASK MANAGER FUNCTION

K_Task_Create

Purpose: This function creates a task and defines it to CMX. The creation identifies to the ROM TCB which ROM TCB task now owns the index slot of the ROM TCB. It also places the task into the IDLE state.

Called

Before entering RTOS and tasks.

#include <cpfuncs.h> /* has function prototype */
byte *K_Task_Create*(byte); /* this is the function prototype */

#define ROM_TCB_ID ??? /* index into the ROM_TCB array, of where this task resides. */

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Create*(ROM_TCB_ID);

Passed

ROM_TCB_ID is the index number into the ROM_TCB array that describes this task.

Returned

CMX returns the status of the *K_Task_Create* function call indicating whether the call was successful or not. If STATUS equals K_OK, then the parameters contained within the ROM TCB are used to set up the ROM TCB array. The task is put in the IDLE state and must be triggered with the *K_Task_Start* function to enable this task to be READY to run.

K_Task_Create example:

void task1(void); /* function prototype, show that task1 does not receive or return parameters */

#define TASK1_ROM_TCB_ID 1 /* task 1 ROM TCB index. */

#define TASK2_ROM_TCB_ID 2 /* task 2 ROM TCB index. */

```
void main(void)
{
   unsigned char STATUS;
   STATUS = K_Task_Create(TASK1_ROM_TCB_ID);
```

```
   /* call the RTOS function K_Task_Create with the index into
   the ROM TCB for task 1.  */
   if (STATUS == K_ERROR)
      {
      /* Error, handle it here */
      }

   STATUS = K_Task_Create(TASK2_ROM_TCB_ID);
   /* call the RTOS function K_Task_Create with the index into
   the ROM TCB for task 2.  */
   if (STATUS == K_ERROR)
      {
      /* Error, handle it here */
      }
}
```

TASK MANAGER FUNCTION

K_Task_End

Purpose: This function allows a task to terminate itself either prematurely or at the end of its code. This function must be called by all tasks that would normally execute their right-end brace. Once called, all variables on the stack will be lost. Make sure other tasks are not waiting on this task. Only the calling task can terminate itself early.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
void *K_Task_End*(void); /* this is the function prototype */

Passed

Nothing is passed. Only tasks can call this function.

Returned

No status is returned.

☞ *An immediate rescheduling will occur when this function is called. If the task has additional trigger (K_Task_Start) requests, then the task will be put into the READY state, otherwise the task will become IDLE. Remember, all local variables will be lost. Other tasks may be waiting for this task to use a CMX function call to wake it. .*

> ✗ **WARNING: ALL tasks must call this function prior to executing their right-end brace. If the task will never, for whatever reason, execute its end brace, then this function does not have to be called and this task is considered compute bound. A task may execute an indefinite while statement, and as long as the while statement will never have a break statement, this function does not need to be present. It is highly recommended that all tasks have this function.**

K_Task_End example:

void task2(void); /* prototype task2 */

```
 void task2(void)
{
   /* task's application code ... */
   ...
}
void task2(void)
```

```
{
   /* task's application code ... */
   K_Task_End();
   /* task 2 has terminated itself early.  Normally this call
   should not be used except in critical error situations, if at
   all */
   ...
   K_Task_End(); /* ALL TASKS MUST HAVE this function, if
   normally would execute their right end brace. */
}
```

Comments

In example 1, task 2 has normally reached its right end brace and MUST execute the *K_Task_End* function to properly terminate itself. In example 2, task 2 has terminated itself early. All locals on stack will be lost. If a task must use this function, it is wise to make sure that no other tasks may be waiting on this task. If a task is waiting, the task about to terminate itself may send a message or flag to another task, so that task may wake the waiting task.

TASK MANAGER FUNCTION

K_Task_Kill

Purpose: This function gives you the ability to remove a task permanently from the RAM TCB task control block queue. The calling task may remove itself, if desired. If the task calling the *K_Task_Kill* function is trying to remove itself, then it will be removed and an immediate task switch will occur. Please note: all tasks waiting on the removed task will not be notified when the task is removed and may wait forever.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
byte *K_Task_Kill*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Kill*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to kill.

Returned

STATUS returned is one of the following:

☞ *Only if the task is not removing itself*

K_OK = Good: function call was successful.

K_ERROR = Error: the task ROM TCB index number does not exist.

If STATUS equals K_OK, then the task has been removed successfully. If the task is removing itself, then an immediate task switch will occur.

K_Task_Kill example:

void task1(void); /* prototype task1 */

#define TASK1_ROM_TCB_ID 0

```
void task2(void)
```

```
{
   unsigned char status; /* local */
   ...
   K_Task_Kill(TASK1_ROM_TCB_ID); /* remove task1, any further
   reference to task1 will be in error */
   if (status != K_OK)
      {
      /* take corrective action. */
      }
   ...
}

void task1(void)
{
   /* declare locals if need be */
   /* task 1 code... */
}
```

TASK MANAGER FUNCTION

K_Task_Start

Purpose: This function takes a task from the IDLE state into the READY state. If the task is in the READY state when this function is called, then no further action is performed.

Called

Before entering RTOS, tasks.

#include <cpfuncs.h> /* has function prototype */
byte *K_Task_Start*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Start*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to start.

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the ROM TCB index number does not exist.

If STATUS equals K_OK, then the task is put into the READY state. If it is in the READY state already, then no further action will be performed.

Again the status is passed back indicating whether the CMX *K_Task_Start* call was successful or not. If the task being started has a higher priority (lower priority number) than the task that is currently RUNNING, then an immediate task switch will occur, bypassing the normal rescheduling caused by the system tick.

K_Task_Start example:

#define TASK1_ROM_TCB_ID 1 /* task1 ROM TCB index number. */

```
void task2(void)
{
```

```
unsigned char status;
status = K_Task_Start(TASK1_ROM_TCB_ID); /* start task 1, put
into READY state if this is the first trigger */
if (status != K_OK) /* check status, make sure good function
call */
   {
   error_handler(); /* go to error handler */
```

TASK MANAGER FUNCTION

K_Task_Wait

Purpose: This function allows a task to suspend itself for a specified period of time or indefinitely. It is very useful for synchronization with time, interrupts, and other tasks.

Called

Tasks.

```
#include <cpfuncs.h> /* has function prototype */
byte K_Task_Wait(word16); /* this is the function prototype */

#define TIME_CNT ???

unsigned char STATUS; /* should be local */

STATUS = K_Task_Wait(TIME_CNT);
```

Passed

TIME_CNT is number of system ticks that this task will suspend itself. If the value is zero then the task will be suspended indefinitely until the *K_Task_Wake* function is used. If the value is non-zero, then the task will be suspended for that number of system ticks. The *K_Task_Wake* function may be used prior to the time left, to wake this function and put it back into the READY state. The maximum value that TIME_CNT may be is 65535.

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task identification number does not exist.

K_TIMEOUT = Warning: the time specified has elapsed.

If STATUS equals K_OK, then the task has RESUMED execution because the *K_Task_Wake* function was used to wake this task. If STATUS equals K_TIMEOUT, then the time period specified has expired and this is why the task was awakened.

> ✗ **WARNING: This function does not test to see if the caller is a task or not, so make sure that only tasks call this function.**

K_Task_Wait example:

void task1(void); /* prototype task1 */

```
void task1(void)
{
   unsigned char status; /* local */
   ...
   status = K_Task_Wait(100);
   /* task 1 will suspend itself for 100 system ticks, unless the
   K_Task_Wake function is used to wake this task prior to time
   period elapsing */
   if (status == K_ERROR)
      {
      /* ERROR, take corrective action. */
      }
   if (status == K_TIMEOUT)
      {
      /* time out occurred.  In some cases this is what you want
      for this task at this time.  If the task was not expecting
      the time period to expire, then corrective action could
      take place here */
      }
   ...
   status = K_Task_Wait(0);
   /* task 1 will now suspend itself indefinitely until the
   K_Task_Wake function is used to wake this task. */
   if (status == K_ERROR)
      {
      /* ERROR, take corrective action. */
      }
   ...
}
```

TASK MANAGER FUNCTION

K_Task_Wake

Purpose: This function wakes a task that had put itself in a suspended state. The task must have used the *K_Task_Wait* function to suspend itself. If the task that "wakes up" has a higher priority than the currently RUNNING task, then an immediate rescheduling (task switch) will occur.

Called

Tasks.

#include <cpfuncs.h> /* has function prototype */
byte *K_Task_Wake*(byte); /* this is the function prototype */

#define TASK1_ROM_TCB_ID ???

unsigned char STATUS; /* should be local */

STATUS = *K_Task_Wake*(TASK1_ROM_TCB_ID);

Passed

TASK1_ROM_TCB_ID is the index number of the ROM TCB indicating which task to wake up.

Returned

STATUS returned is one of the following:

K_OK = Good: function call was successful.

K_ERROR = Error: the task ROM TCB index number does not exist.

K_NOT_WAITING = Error: the task specified was not waiting.

If STATUS equals K_OK, then the task specified has been "awakened" and put into the READY state. If STATUS equals K_NOT_WAITING, then the task specified was not waiting for this function call.

K_Task_Wake example:

#define TASK1_ROM_TCB_ID 0

```
void task2(void)
{
   unsigned char status;
```

```
   status = K_Task_Wake(TASK!_ROM_TCB_ID); /* wake task 1 up */
   if (status != K_OK) /* check status, make sure task was
   waiting */
      {
      /* maybe take corrective action if task 1 wasn't waiting */
      }
}
```

C

Clear mode command 21
CMX Multi-Tasking Executive 1
CMX Scheduler Chapter 44
CMX Scheduler task control blocks 6

E

Event manager function 20
Event Manager Functions
    K_Event_Reset 26
    K_Event_Signal 23
    K_Event_Wait 20

G

Getting started 2

I

IDLE 5, 44
Interrupts 4, 8

K

K_Event_Reset 26, 48
K_Event_Signal 23, 50
K_Event_Wait 20, 52
K_OS_Disable_Interrupts 33, 54
K_OS_Enable_Interrupts 32, 56
K_OS_Init 28, 57
K_OS_Intrp_Entry 29, 58
K_OS_Intrp_Exit 30, 59
K_OS_Start 28, 60
K_OS_Tick_Update 30, 61
K_Task_Coop_Sched 16, 63
K_Task_Create 10, 64
K_Task_End 17, 66
K_Task_Kill 18, 68
K_Task_Start 12, 70
K_Task_Wait 13, 72
K_Task_Wake 15, 74
KILL 5, 45

M

Mode values in the K_Event_Signal function 23

O

Operating System Functions 27
    K_OS_Disable_Interrupts 33
    K_OS_Enable_Interrupts 32
    K_OS_Init 28
    K_OS_Intrp_Entry 29
    K_OS_Intrp_Exit 30
    K_OS_Start 28
    K_OS_Tick_Update 30

P

Preemption 4, 44
Priority 4

R

RAM TCB 6
READY 5, 44
RESUME 5, 44
Return Status Byte Values 7, 47
    K_ERROR 7, 47
    K_NOT_WAITING 7, 47
    K_OK 7, 47
    K_TIMEOUT 7, 47
ROM TCB 6
RUN 5, 44

S

Scheduler 4
Setting Up Tasks 6
Stacks 38
Synchronization 23, 24
System tick 4

T

Task Manager Functions 9

K_Task_Coop_Sched 16
K_Task_Create 10
K_Task_End 17
K_Task_Kill 18
K_Task_Start 12
K_Task_Wait 13
K_Task_Wake 15
Task states 5
IDLE 5, 44
KILL 5, 45
READY 5, 44
RESUME 5, 44
RUN 5, 44
WAIT 5, 44
Task switch 4

W

WAIT 5, 44
WAIT (suspended) state 5, 44
When A Task Is Interrupted 4